Online Spectrum-based Fault Localization for Health Monitoring and Fault Recovery of Self-Adaptive Systems

Éric Piel*, Alberto Gonzalez-Sanchez*, Hans-Gerhard Gross* Arjan J.C. van Gemund*, and Rui Abreu†
*Department of Software Technology
Delft University of Technology
Mekelweg 4, 2628CD Delft, The Netherlands
{e.a.b.piel, a.gonzalezsanchez, h.g.gross, a.j.c.vangemund}@tudelft.nl
†Department of Informatics Engieering
Faculty of Engineering of University of Porto
Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
rui@computer.org

***Abstract*—Software systems used in the industry are often large and complex. Even with an extensive validation phase, it is impossible to ensure that a software system is fault-free and will remain so all along its evolution. When a failure happens in operation, the time to solve the fault should be minimized. The major challenge in this realm is the localization of a fault in one of the constituent components of the overall system. We strive at simplifying the localization of the fault that led to a failure by adapting existing techniques to the online context in such a way that allows the system to be aware of its own internal faults and react to it. This article first proposes to apply the Spectrum-based Fault Localization (SFL) method for online fault localization and health monitoring. Several implementation approaches are presented with a performance that depends on the architecture and the framework used. Evaluation is done through simulation of online failure scenarios, and through implementation in a demonstration surveillance system. The results of the studies performed confirm that applying SFL online, using monitoring, can successfully provide health information and locate problematic components, so that a software failure can be addressed adequately and timely.**

Keywords-Fault localization; diagnosis; self-awareness; autonomous system; monitoring; component-based system.

I. INTRODUCTION

It is generally accepted that all but the most trivial software systems will inevitably contain residual defects. Large and complex software systems, such as systems of systems, will face these problems. Nowadays, the high reliability, availability, and flexibility imposed on many systems require support for online reconfiguration and join/leave of external components (a coupled and cohesive part of a system). This further increases the chances of unexpected behavior during execution, as they are hard to take into account in the validation phase. As such problems cannot be avoided, the system should be prepared to handle them as quickly as possible. Typically, after a failure (a deviation from the expected behavior) has been detected the following steps are taken: diagnosis, bug fix design, re-validation, and update. To reduce the time of this process, we focus here on automating the diagnosis step, which very few previous works in adaptive systems have tried to automate. This step focuses on finding the location of the fault, i.e., the cause of one or more failures in the system.

So far automated diagnosis techniques, also called fault localization, have been applied solely offline, during the testing phase. In this article, we detail approaches to apply fault localization in an online context, i.e., when the system is in operation. One of the obstacles is that typical *active testing* used offline cannot be applied online, because of interference with the normal operations. So continuous validation must come from observations provided by monitors, also referred to as *passive testing*. While there exist other approaches to fault localization [1], [2], [3], SFL is one of the most light-weight fault localization techniques available to be used for the provision of health information and for identifying problematic components in software systems.

In this paper, we make the following three contributions. (1) We demonstrate how SFL can be applied to online fault localization by introducing three main adaptations to the original technique. (2) We describe two different approaches for the implementation of online fault localization according to the characteristics of the software system. (3) We assess the performance of our proposed techniques in simulations as well as in a real industrial case study.

The original SFL technique is described in Section II. Section III presents the modeling of the problem. Our proposal of online fault localization is presented in Section IV. Section V summarizes the main approaches we have used to implement fault localization on actual software systems. Section VI evaluates the technique on a case study. Finally, Section VII discusses related work, and Section VIII concludes the article.

II. SPECTRUM-BASED FAULT LOCALIZATION

The objective of fault localization is to pinpoint the precise locations of faults in a system. Before delving into

 ISBN: 978-1-61208-187-8

$\mathcal{C}$	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	SC
	`function count(char *s) {`							
	`  int let, dig, other, i;`	0	0	0	0	0	0	0
c_0	`  let = dig = other = i = 0;`	1	1	1	1	1	1	0.87
c_1	`  while (c = s[i++]) {`	1	1	1	1	1	1	0.87
c_2	`   if('A'<=c && 'Z'>=c)`	1	1	1	1	0	1	0.93
c_3	`    let += 2;`	1	1	1	1	0	0	1.0
c_4	`   else if('a'<=c && 'z'>=c)`	1	1	1	1	0	1	0.93
c_5	`    let += 1;`	1	1	0	0	0	0	0.71
c_6	`   else if('0'<=c && '9'>=c)`	1	1	1	1	0	1	0.93
c_7	`    dig += 1;`	0	1	0	1	0	0	0.71
c_8	`   else if(isprint(c))`	1	0	1	0	0	1	0.47
c_9	`    other += 1;}`	1	0	1	0	0	1	0.47
c_{10}	`  printf("%d %d %d\n",`	1	1	1	1	1	1	0.87
	`          let, dig, other);}`							
	Test case outcomes	1	1	1	1	0	0	

Table I
EXAMPLE PROGRAM, SPECTRUM, AND OUTPUT IN SFL.

the usage of the SFL approach for online fault localization, and the provision of health information, let us introduce SFL in its offline version.

The following data are usually used as inputs in SFL approaches:

- A finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_j, \ldots, c_M\}$ of M *components* (e.g., source code statements, functions, classes) which are potentially faulty. We will denote the number of faulty components in the system as M_f.
- A finite set $\mathcal{T} = \{t_1, t_2, \ldots, t_i, \ldots, t_N\}$ of N given tests with binary outcomes $O = (o_1, o_2, \ldots, o_i, \ldots, o_N)$, where $o_i = 1$ if test t_i failed, and $o_i = 0$ otherwise.
- An $N \times M$ *coverage matrix*, $A = [a_{ij}]$, where $a_{ij} = 1$ if test t_i involves (covers) component c_j, and 0 otherwise. Each row a_i of the matrix is called a *spectrum*.

Table I shows an example of SFL applied on a small program with a component granularity at the statement level. This program aims at counting different types of characters. The component c_3 contains a fault, mishandling uppercase characters. 6 tests are executed against this implementation. The columns t_1 to t_6 present the coverage spectrum and the test outcomes when executing each of the tests. The last column shows the similarity coefficients, a value computed by the SFL, which we will describe later.

The output of fault localization is a *diagnosis*, which is a ranking of the components ordered according to their assumed likelihood to contain a fault.

In program debugging, the granularity of a *component* is often very small, typically at the statement level, since SFL benefits from variations in program control flow (i.e., different branches of a `if` are taken). However, in an online context, a larger grain size for components is more appropriate. This still permits to monitor a system and to take the appropriate actions in case of degradation, while it reduces the performance overhead, and represents a more realistic component granularity for large systems. In the later study, we selected a granularity at the level of the source code functions.

A. Statistical Spectrum-Based Fault Localization

Statistical SFL is a well-known approach originating in software engineering [4], [5], [6]. Fault likelihood l_j (and thus assumed health) is quantified in terms of *similarity coefficients*. Intuitively, the goal is to identify the component whose line of test coverage is most similar to the test outcomes. Similarity coefficients measure the statistical similarity between component c_j's test coverage $(a_{1j}, \ldots, a_{Nj})$ and the observed test outcomes, $(o_1, \ldots, o_N)$. It is computed by four values $n_{pq}(j)$ counting the number of times a_{ij} and o_i form the combinations $(0,0), (0,1), (1,0), (1,1)$, respectively, i.e.,

$$n_{pq}(j) = |\{i : a_{ij} = p \wedge o_i = q\}| \;\; p, q \in \{0, 1\} \qquad (1)$$

For instance, $n_{10}(j)$ and $n_{11}(j)$ are the number of tests in which component c_j is executed, and which passed or failed, respectively. For each component, the four counters sum up to the number of tests N. There are several different known similarity coefficients which are efficient. For example, Tarantula [5], and Ochiai [4] are both very common similarity coefficients. We use the latter one, given by

$$\text{Ochiai:} \quad SC = \frac{n_{11}(j)}{\sqrt{(n_{11}(j)+n_{01}(j)) \cdot (n_{11}(j)+n_{10}(j))}} \qquad (2)$$

Ordering the components by their similarity coefficients results in the ranking of the diagnosis algorithm.

In Table I, the similarity coefficient for each component is indicated. As c_3 was the part most used when a test failed and less used when a test passed, its similarity coefficient is the highest. The SFL will therefore rank c_3 as the most likely location of the fault, which is correct.

A by-product of statistical SFL is the *component health*. The health of a given component can be simply approximated by $h = 1 - SC$, where SC is the similarity coefficient. This permits the system, or system of systems, to also be self-adapting to the failures. Components which have access to redundant information can adapt the weight of each input depending on the health of the components that provide it. For example, in the maritime safety and security context, when a radar starts behaving incorrectly, the situation awareness component can reduce automatically the importance of the data from this radar in its computations.

Despite their lower diagnostic accuracy [7], similarity coefficients have a ultra-low computational complexity (compared with probabilistic diagnosis approaches, such as Bayesian reasoning [5]), which is ideal for online diagnosis. Another advantage is the fact that statistical SFL is incremental. Only the counters n_{pq} must be kept per component, so there is no need to compile a (possibly huge) test coverage matrix. Finally, unlike other approaches, statistical SFL is robust with respect to uncertainties in the test outcomes.

 ISBN: 978-1-61208-187-8

While all techniques tolerate false negatives (i.e., a test involving a faulty component and not returning a failure), statistical approaches are more robust with respect to false positives (i.e., a test reports a failure although the system actually behaved correctly), which is essential in online monitoring as the oracles are often less sophisticated than in offline testing.

B. Diagnosis Effort

In order to compare different diagnosis approaches, there is a need to measure how well a diagnosis performed. This measure, the diagnostic performance, should represent how well the diagnosis algorithm can pinpoint the true root cause of an observed problem. In software fault localization, this performance is often expressed in terms of a metric C_d that measures the theoretical *effort* still needed for a diagnostician to find all faulty components after reading the generated diagnosis [7]. C_d is expressed as the position of the last faulty component in the ranking given by the fault localization. C_d measures *wasted effort*, independent of the number of faulty components M_f in the system, to enable an unbiased evaluation of the effect of M_f on C_d. Thus, regardless of M_f, $C_d = 0$ represents an ideal diagnosis technique (all M_f faulty components are ranked at the top, and no effort is wasted for a human to check healthy components), while $C_d = M - M_f$ represents the worst diagnosis technique (checking all $M - M_f$ healthy components before the M_f faulty ones), with M the total number of components. For example, consider a diagnosis algorithm that returned the ranking $\langle c_{12}, c_5, c_6, \ldots \rangle$, while c_6 contains the actual fault. This diagnosis leads the developer to inspecting c_{12} and c_5 first. As both components are healthy, C_d is increased by 2, and the next component to be inspected is c_6. As it is faulty, no more effort is wasted and $C_d = 2$. To ease comparison between systems, a *relative wasted effort* is often used: $\frac{C_d}{M-M_f}$. A perfect diagnosis gives therefore a relative effort of 0, while the worse possible one gives an effort of 1, and an algorithm picking randomly a component gives on average a relative effort of 0.5.

III. Simulation of a Faulty System

For initial illustration and evaluation of online SFL we use synthetic system simulations next to an actual case study. The main advantage of the simulations is that they can be executed quickly (e.g., for our case study system we can simulate one hour of operation in just a few seconds). They allow to vary many properties of a base system, in order to generalize the findings according to many different (synthetic) system configurations, and they also avoid implementation details which could cause noise in the observations (e.g., test outcomes with false positives).

A. System Model

The simulations use system models with different topologies all based on the surveillance system used as case study, which is presented in Section VI. The simulation of a system generates outputs similar to the ones given by the actual SFL algorithm, i.e., a ranking of the components according to their assumed health over the whole period of execution of the simulation. The simulator and example models are available for download [8].

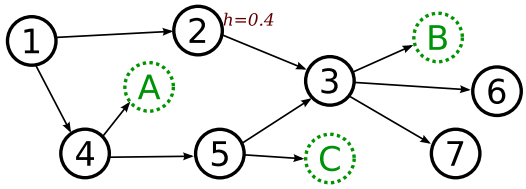

Figure 1. Example topological layer with 7 functional components and 3 monitors.

Fig. 1 shows an example of a system model, with 7 functional components and 3 monitors (A, B, and C). As we will see in Section IV, monitors are placed in order to replace test cases in an online context. Component 2 is set to be faulty, with a fault happening 60% of the time it is used. The model represents a typical data-flow system where component 1 receives the inputs and passes them on to the other components. More information about the simulation setup and a description of the type of model that is used in the simulator can be found in [9].

B. Simulated System Generation

One of the most difficult parts of simulation is to obtain models of systems which are representative of the reality. If a model is generated fully randomly with respect to every possible parameter, there is little chance that it corresponds to a potential real system. That is because only some topologies, order of execution, etc. are reasonable for a software system. Therefore, as basis for creating many simulations, we used the topology of a known surveillance system. It comprises 63 components for the functionality. For each component, a configuration was generated with that component being the faulty one. For each fault location, 10 different system configurations were generated by randomly placing 15 monitors, and producing a set of 20 execution paths (with random frequencies between 0.2 Hz and 50 Hz). Therefore, each technique can be evaluated on 630 system configurations. Results are presented in the next section.

IV. Online Fault Localization

Applying SFL online brings up three issues: (1) test cases would disrupt the normal operation of the system (to be discussed in Section IV-A, (2) the range of a coverage spectrum (to be discussed in Section IV-B), (3) the adequacy of the diagnosis with the current system behavior (to be discussed in Section IV-C). In an offline context, tests are run separately, so the start and end of a test and the coverage spectrum are clear, as well as associated inputs

 ISBN: 978-1-61208-187-8

and outputs. However, in the case of continuous diagnosis these boundaries disappear, or, at least, become blurred. In this section, we present solutions for adapting SFL to an online context.

A. Obtaining Test Outcomes Online

In order to bring fault localization online, the usage of test cases must be reevaluated. Test cases are *active*, as they provides their own inputs to the system. If done during operation, such input can interfere with the usual behavior of the system, and can cause a large performance overhead. Therefore, in the online context, *monitoring* is more fitting. Monitoring is well-understood, easy to apply, and event-based, due to its passive nature, e.g., triggered by the arrival of new data, or a timer interrupt. A monitor is a specific component in the system that observes and assesses the correctness of the functionality without interfering through test inputs.

A monitor observes data or behavior at specific locations and decides based on built-in oracle logic whether an observation is expected (*pass*) or unexpected (*fail*), for example through checking the range of a variable, consistency between different data, or through comparison with a state model. The monitor outcomes replace the test outcome. Because SFL requires to know when the system is deemed behaving both correctly and incorrectly, it is of prime importance when writing a monitor that whenever a *fail* could be sent, it sends a *pass* if no failure is detected.

B. Spectrum Sampling

In many cases, interactions in a live system are not clearly separable by time or space boundaries (such as a complete test transaction in testing). Input stimuli are continuously arriving and the system responds accordingly changing its internal state and/or producing some output. For example, in our case study (cf Section VI), input messages arrive at any time, and sometimes simultaneously in separate threads. Previous inputs influence the behavior of a component either explicitly such as in a database, or implicitly by affecting its internal state. When applying SFL offline, the coverage spectrum is recorded since the system was started for a test case. In an online context, after a short period of operation, the coverage matrix will contain only 1's: "everything covered". Although this approach would guarantee a theoretical strong causal relationship between fault execution and failure observation (i.e., if a failure is observed, the spectrum will contain the fault information), a solid 1's spectrum does not provide any diagnostic information for the SFL, because it infers the diagnosis from differences of the various spectra in the coverage matrix A and the outcome O. The curve named *time inf* of Fig. 2 shows the result of never resetting the spectrum. The average diagnostic cost is approximately 0.5 all the time. Guessing the fault locations randomly would yield a similar performance.

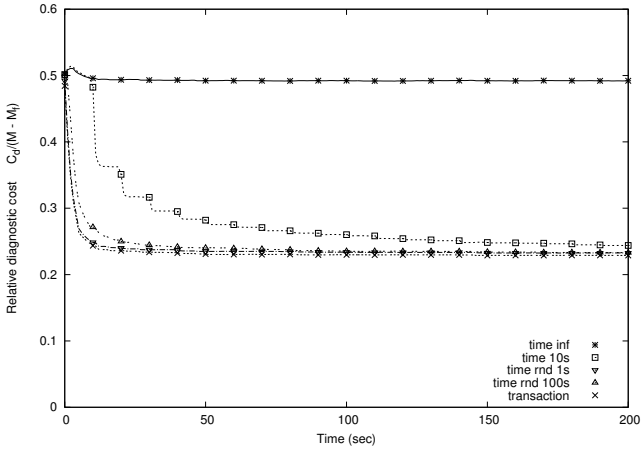

Figure 2. Average diagnostic cost along the time of observation for various observation policies, with simulated systems having one fault.

The coverage of components represented as binary values in the spectrum must be reset regularly, in order to provide a meaningful diagnosis. We propose two different solutions, which are adapted to different development contexts, that is, a transactional approach, and a time frame approach.

1) Transactional: A monitor validates the correctness of a specific component transaction in the system, corresponding to particular interactive functionality. The provision of an outcome through the monitor correlates to the end of this transaction. The *transactional* policy assigns a separate spectrum to every monitor. Every monitor is also associated to a scope, which represents which components might be involved in the monitored interaction[1]. Each time a component is involved, the current spectrum of every monitor whose scope contains that component is updated. When a monitor generates an outcome, its associated spectrum is used as a row for the matrix A and is then completely reset to zero.

The list of the components in the scope associated to each monitor is provided before the start of the system (and is updated after each modification). It is either manually created by the user (the developer of the monitor, most likely), or it could be determined by code or configuration analysis. Fig. 2 shows with the curve *transaction* that this solution is the most effective one, with a low average diagnostic cost throughout the execution of the systems. The curve tends towards an asymptote close from 0.2. This asymptote corresponds to the average diagnostic cost that can be achieved by the SFL algorithm with all possible spectra for the specific set of systems in the simulation.

However, if a fault modifies how components interact (i.e., the control flow is modified), the difference between the expected behavior and the implementation could lead to an inaccurate scope. In such a case, this policy would

[1]Each execution of the interaction can be considered a *transaction*, hence the name of the policy.

 ISBN: 978-1-61208-187-8

cause a faulty component to be omitted from every spectrum associated with a *fail* outcome. The quality of the diagnosis would be adversely affected. In addition, pre-analysis of the system for every monitor can be time consuming, and needs to be done every time the system is modified. It might be difficult to perform if external components (from different companies) are used. In order to avoid this analysis we investigate a technique requiring less information about the system, i.e., the *time frame* technique.

2) Time Frame: The *time frame* policy uses expiration of time as transaction boundary to establish causality between components covered and monitor outcome. Over a given time period, the component activity is recorded into a global "current spectrum". When the time expires, the bits of the involved components are reset and the recording of a new current spectrum is started. Every monitor outcome during this period, is associated with the current spectrum.

Time frame-based sampling avoids spectra with too many 1's if the time window is properly adjusted to the working speed of the system. To avoid using a period which could hide a specific fault our approach uses a random frame length. After expiration of a time frame, the length of the next frame is determined randomly within reasonable bounds. An exponential distribution is used, in order to have a broad set of period sizes. An average period must be selected according to the system under observation, but it can be relatively roughly estimated to the average processing time of a typical transaction. In Fig. 2 it can be seen how a fixed time period leads a limited accuracy of the fault localization, with the curve *time 10s*. The curves *time rnd 1s* and *time rnd 100s*, corresponding respectively to a randomized time frame with an average of 1 s and 100 s, both provide on average a low diagnostic cost.

We recommend that the observation policy should be selected according to the system context: if it is possible to gather precise information on which interaction is observed by a monitor, then the transactional policy should be applied. Otherwise the randomized time frame policy should be implemented, with just enough validation to ensure the average period is adapted to the system.

C. Spectrum Matrix Size

When using SFL offline, the size of the spectrum matrix and the test outcome vector are finite and, in practice, relatively small, which is not the case online. For example, in our case study, approximately 100,000 monitor outcomes are generated for a single hour of observation. This could eventually lead to excessive storage requirements and processing overheads. This potential size problem is addressed through application of statistical SFL, on which our approach relies. It is incremental, so that accumulating counters can be used.

However, another issue is the timeliness of a spectrum, for example "is a week-old observation relevant for the current state of the system?" A fault may appear long time after the system was started (e.g., memory leakage, unexpected combination of inputs that affect the internal state of the system, an unnoticed third-party component update). Old spectra might mislead the fault localization. The detection of a new failure should always lead to the same diagnosis, independent of how long the system has been running.

Note however that the problem is not symmetric, when conversely, a fault is fixed, or the failures are not observed anymore. If the fault is fixed, it is easy to reset the matrix at the same time to avoid this "aging effect". If the failures stop appearing without the fault having been fixed, it is better to still report the component as faulty for some sufficiently long time to acknowledge the problem and deal with it.

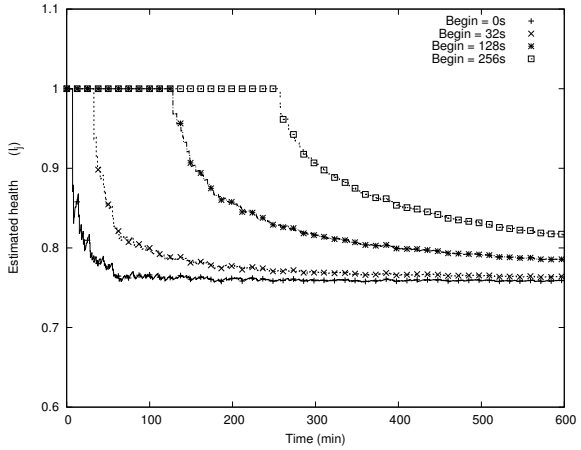

Figure 3. Estimated health with an infinite window.

Fig. 3 shows the health estimated by the SFL algorithm for a faulty component yielding a failure at different times, when all spectra are kept. The later the failure surfaces, the slower is the convergence of health. From the point of view of the system maintainer, when a given failure happens, the algorithm output should be identical independently from the time system has been running previously.

To overcome this problem, we defined the *sliding window* policy. Spectra that are older than a given age are discarded. In practice, as the SFL counters are accumulated, we approximate the window by decomposing it into a fixed number of small periods. An array of counters allows to keep track of the SFL counters for each period. When the current period is over, the oldest set of counters is discarded and replaced by a new set for the next coming period. The global counters are replaced by an addition of the counters for each available period. In our implementation we used 32 sub-periods, which appeared to be of sufficient precision.

The ideal window size (leading to stable health values) depends on the frequency of the monitors generating observations and the frequency of failures being detected. In our experiments, we observed that short sliding windows yield a relatively high diagnostic cost and unstable output

 ISBN: 978-1-61208-187-8

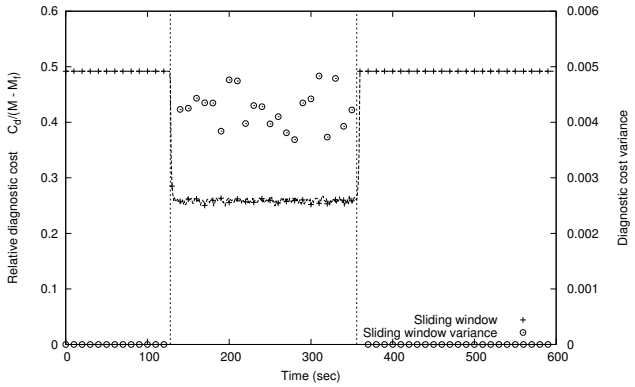

Figure 4. Average diagnostic cost on simulated systems with a sliding window policy of length of 4 s (component fails in the period 128 s – 356 s, dotted lines).

over time, because they are too small to contain enough test outcomes for adequate diagnosis. When the size of the window is extended, it reaches a point where the diagnostic performance does not improve anymore. Increasing further the length solely leads to a bigger latency to react to the failure disappearance. The size of the window after which the diagnosis presents no more noise depends on the frequency at which the failures are detected. We observed that the minimum efficient window size depends on the amount of *fail* outcomes that are captured. The amount of *pass* outcomes is usually far superior, so it is not a bottleneck. We observed that if a window is long enough to contain at least approximately 10 *fail* outcomes, it is sufficient to keep a good quality diagnosis.

Therefore, we recommend selecting a size of the window which is sufficiently long to receive many monitor outcomes. The main restriction on the maximum length is to ensure a fairly fast reaction in terms of health. The window size can be set as the minimum duration for which a single failure occurrence should be seen when looking at the diagnosis.

In order to observe the effect of applying the sliding window policy, we simulate a system where a new failure is seen, lasts for 228 s, and disappears. Fig. 4 shows the average diagnostic cost when a window size of 4 s is applied. Approximately 4 seconds after the first failure appears the diagnostic cost reaches its minimum. Similarly, the diagnostic reacts within seconds to the disappearance of the failures. As the failure frequency is high enough that a window contains several *fail* observation outcomes, the diagnostic variance is relatively low. Increasing the window size would stabilize even further the diagnostic over the period that the failure happens.

D. Self-Adaptation to Faults

By localizing properly and precisely the faults, a system has two main ways to react in order to improve its behavior. Firstly, it can attempt to fix the failure origin by applying an automated fix such as described in [10]. Such automated fixes rely usually on a set of generic fixes. Each of the generic fix can be apply sequentially after each other, on the each of the most suspicious fault location provided by the online SFL. The search for a fix ends when the online SFL detects that the health of the system goes back to an acceptable level (or when all the fixes have been tried unsuccessfully).

A second way to adapt to a fault, orthogonal to the first one, is to take into account the estimated health of the components into the functional behavior. As seen previously, SFL computes for each component a *similarity coefficient*, which can be converted into an estimated health value approximating how likely the component provides a correct output. The *confidence* of a data is the product of the health of each component which was involved in generating it. In dependable systems, it is usual to obtain data from multiple independent sources and/or process the data via redundant paths. Instead of relying equally on all the redundant data, components which receive data from multiple sources can weight the data according to their confidence value. Therefor, the system adapts automatically to faults by avoiding to rely on the incorrect data.

V. Implementation of Online Fault Localization

There are many ways to implement the proposed techniques. We outline here two different implementation approaches that we have carried out successfully. The first approach is centralized, while the second one is metadata-based.

A. Centralized Approach

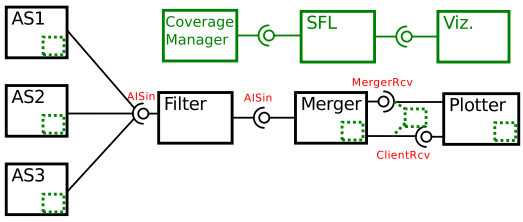

Figure 5. Architecture of the case study system, which is based on the centralized approach.

A first implementation approach, which we have used in our Atlas framework [11], relies on a centralized spectrum recording. Its architecture can be broken down into five parts. An example system using such an approach is displayed in Fig. 5. For each architectural part, we will refer to this example. The *coverage manager* component takes care of keeping the coverage spectrum of the system. In the example, this component is represented by the box of the same name. The spectrum is reset periodically according to the randomized time frame policy as described previously. By request from the coverage instrumentation part (discussed later), it sets a position in the spectrum to indicate a specific component

 ISBN: 978-1-61208-187-8

has been covered. When a monitor sends a new observation, the coverage manager receives this observation, attaches the current spectrum, and forwards it to the SFL component.

The *SFL* component (which is represented by the *SFL* box in the example) receives every monitor observation and adds it to the matrix according to the sliding window policy. In practice, a whole matrix is not needed, only a set of accumulators, which permits a fast processing. Running at a slower frequency, the similarity coefficient and ranking of the faulty components is computed. This might require a noticeable amount of processing power, but it can be done independently from the rest of the system, even offloaded to separate hardware.

Every functional component of the system is instrumented to report whenever one of its methods is called. In the example, every component part of the core functionality is instrumented. We use Aspect Orientation [12] and Java self-reflection to apply the same code to all the components. This allows to dynamically instrument any component, even when provided by a third party or added a posteriori. However, it brings a high overhead to each method call. A static approach, such as found in many code profilers, would likely be more efficient.

Finally, the behavior of the system is validated by a set of monitors, positioned at various places between or around the normal components. Monitors are represented as dash boxes in the example. Every monitor observation, both *fails* and *passes*, is transmitted to the *coverage manager*. A monitor can be replacing what would traditionally be a warning or error check, or can be more complex piece of code which validates the outputs of a component compared to the previously received input (based for instance on a state machine). Watchdogs, which detect the loss of service provided by a component can also be implemented as monitors but care should be taken to report in case of failure not the actual spectrum, but the spectrum that would be expected (so that SFL can point towards the non-responding part of the system).

Last but not least, the *visualization* component receives the measurements from the *SFL* component and displays to the user a graph of the health of the components (approximated by their similarity coefficient) over time.

B. Metadata-Based Approach

The centralized approach is easy to implement and efficient on systems where all components can access the *coverage manager* with a low latency and where communications have a low overhead. In systems where components are running on physically separate nodes such as systems-of-systems, or systems which are message-based, it might be more efficient to use a different approach, based on metadata. All data transmitted between components is associated to metadata that contains a coverage spectrum indicating all the components used to generate this data. Every time an output is generated, its metadata must be set, based on the metadata of the inputs. Note that computing the spectrum might be difficult in some cases where many inputs are used. There is still a central component for the coverage, but it is only accessed to request a position in the spectrum when a component initializes. Monitors work similarly to the previous implementation approach except that the spectrum associated to an observation comes from the metadata of the output which is validated. This observation can then be sent directly to the SFL component.

To handle dynamic system architectures, where components can be added and removed online, the coverage spectrum needs to have positions updated when there is a change. We treat this requirement by having the *coverage manager* assign positions to new components. When a component is removed the positions which were assigned to it can be reused, once a certain delay corresponding to the time window length has passed.

VI. Case Study

All techniques for realizing online fault localization with SFL have been introduced. Synthetic system simulations were used to compare different techniques to each other on a large set of systems. In the following, we evaluate our contributions on a real system. The main goal is to validate the techniques on practical ground, and verify that the simulated systems behave similarly to the actual ones.

The surveillance system that we use as case receives information broadcasts from ships, called *AIS messages* [13], and it processes them in order to form a situational picture of a maritime area. The system is made of Atlas components in Java. In total it is comprised of 63 methods (the granularity of the SFL) for the core functionality with an average of 10 lines of Java code each.

The monitoring infrastructure comprises four monitors, each of them guarding different functional and non-functional aspects of the system. Coverage of components is recorded through an ad-hoc Java aspect, as described in Section V.

A. Injected Faults

We simulate two types of faults, loss of data between components (for example due to reset of the component, or unstable connection), and software faults caused by the functionality. Data loss faults are simulated through intermittent connection drops between two components. Software faults are introduced through mutations in the original code (a set of 100 mutants which was created with μJava [14] and manually verified to affect the behavior of the system). For each of the mutation faults, the system was executed for one hour with the recorded input, producing approximately 100,000 monitor outcomes in total. A posteriori, it is then possible to determine the diagnostic cost at each moment in time. 12 mutations lead to early system crash (within a

 ISBN: 978-1-61208-187-8

minute) and are sorted out (in practice, such a bug would be directly noticed and investigated off-line). 55 mutations have faults not detected by the monitors, leaving 33 configurations with detectable faults.

B. Results

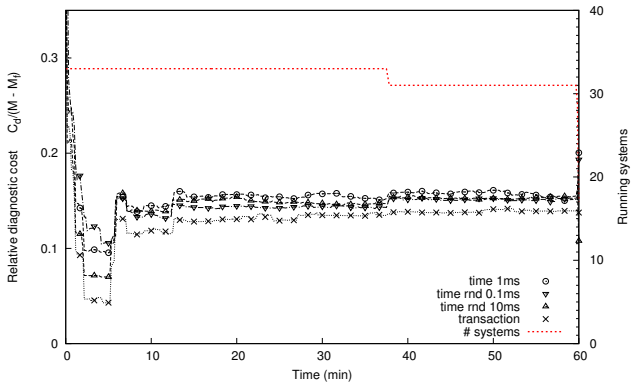

Figure 6. Average diagnostic cost (33 configurations) over the time for three different observation policies.

The average C_d for *transactional* and *randomized time frame* observation strategies is presented in Fig. 6. The *# systems* indicate the number of systems still running at a given time. It decreases whenever a system crashes or stop responding. The SFL algorithm uses a sliding window of 5 minutes, in order to ensure a good quality of the diagnosis while keeping a relatively fast reaction to any fault correction.

The diagnostic cost C_d, which starts at 0.5, decreases until it reaches some relatively constant value after around a minute. This is similar to the results seen in the simulations (Fig. 2). After 5 minutes of execution (i.e., the length of the sliding window), all C_d graphs increase. This is because some faults lead to failures only at initialization, i.e., they are located in components only used at that time. When these first spectra are removed from the matrix (through the sliding window) the SFL loses information about their location, and assumes a better health, leading typically to a $C_d = 0.5$. Hence, the average C_d increases.

As in the simulation, the *transactional* observation performs best, with an average $C_d = 0.14$. The *time frame* observation yields its best results with 1 ms ($C_d = 0.16$). A shorter or longer period impairs the results, leading to C_d around 0.3 (not shown in the figure to improve readability). This suggests that observation periods of 1 ms are optimal for this system. The *randomized time frame* observation performed equally well as the best fixed time period, for all periods tried between 0.1 ms and 100 ms.

In our case, transactional observation provides the best results. Nevertheless, this requires that for each monitor the information about which components are observed is known and correct. Otherwise, a randomized time frame allows diagnosis with comparable quality, with only a rough estimation of the processing time needed.

This case study demonstrates the feasibility of online fault localization using the SFL technique in a system inspired by industry. With a diagnostic cost ranging on average below 0.2 just after a minute, it also shows that fault localization is able to point into the right direction for identifying problematic components in software systems. Of course, this works only if residual defects can be detected by the monitors. The fact that the results are relatively similar to the results obtained by simulation suggests that the model employed for the simulation is representative of this real case.

The case study shows also that a relatively small number of monitors (compared to the number of components) is sufficient to locate faults. Although no complete study has yet be done on the needed number of monitors for a given system, our first observations are that 1) this can vary considerably depending on the topology of the system and the false negative rate of the faults, and 2) for a system of N components, a large number of fault locations can be correctly found when the number of monitors is above $log(N)$.

VII. Related Work

The role of fault diagnosis for realising more adaptive, intelligent, and self-aware systems has been recognized for at least a decade (e.g., [15], [16]). Some researchers have looked at online defect detection [17], [18], but did not address the specific issues of finding the root causes of defects, i.e., the diagnosis.

Seltzer and Small [19] and Chen [20] have proposed system infrastructures for enabling self-monitoring and -adaptation. However, their approaches focus on system performance, ignoring all the other software quality issues, that our approach is able to treat. The biggest drawbacks of these approaches is that they rely on ad-hoc localization algorithms, which are based on long observations performed in test systems rather than in the operational systems, and that they often require manual adjustments. The usage of automatic diagnosis in our approach avoids these drawbacks. Our approach can be applied in a generic way, and relies only on the latest observations.

In [2], an invariant-based approach is presented and applied online. However, they use specialized active unit-testing instead of monitoring, and the system state is recorded every time a test is executed, which leads to a very high overhead (execution time multiplied by $\sim$100). An additional issue are interferences that active testing can cause in a running system.

In [21], an approach for self-repair, coined Rainbow, which allocates the diagnosis process to the individual repair

 ISBN: 978-1-61208-187-8

handlers is presented. Rainbow defines a set of repair strategies that are triggered when certain architectural invariants are violated in a running system. Thus, each strategy is responsible to realize (i.e., diagnose) whether or not its set of actions will fix the observed problem.

In [22], an approach for architecture-based run-time fault diagnosis is presented. Conversely to our approach, the one presented in [22] applies a lightweight model-based approach to fault diagnosis based on the architecture description of the system at the granularity level defined by the architecture (typically, coarse granularity). Similar to the Rainbow approach, pass/fail information is obtained by checking whether architectural invariants are violated in a running system or not.

VIII. Conclusions and Future Work

While fault localization is a fundamental step towards adaptive and self-managing systems, in order to identify the part of the system which should be corrected, little work so far has focused on adopting existing diagnosis approaches into this domain. In this article, we present an approach for realizing online spectrum-based fault localization to be used in self-adaptive systems. We introduce techniques to obtain a significant spectrum for the SFL algorithm in order to yield good diagnoses. The usage of a sliding window, provides a diagnostic outcome which is always relevant to the current state of the system. Furthermore, we presented two different implementation approaches which fit either to centralized architectures or distributed architectures.

Our contributions are validated first by simulation of a large set of randomly generated systems, and through a case study with a system inspired by industry. The diagnostic results on a set of real, mutated systems corroborate the results of the simulation and confirm that, with our contributions, SFL and monitoring can be applied successfully to online fault localization.

Additional challenges could be investigated in future work in order to improve the quality of online fault localization in real systems. One of the main topics we will investigate is the usage of runtime testing to complement the monitors. When a fault is detected but its location cannot be precisely pinpointed, a small set of runtime tests could be executed on the system in order to obtain more information.

Acknowledgement

This work is part of the ESI Poseidon project, partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, May 2005, pp. 342–351.

[2] D. Slane, "Fault localization in in vivo software testing," Master's thesis, Bard College, Massachusetts, USA, 2009.

[3] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," in *IEEE Special Issue on Modeling and Design of Embedded Software*, 2003, pp. 212–237.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Windsor, UK: IEEE Computer Society, Aug. 2007, pp. 89–98.

[5] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. Orlando, FL, USA: ACM, May 2002, pp. 467–477.

[6] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. van Gemund, *Spectrum-based fault localization in practice*. Eindhoven, The Netherlands: Embedded Systems Institute, 2009, ch. 10, pp. 113–124.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *24th IEEE/ACM International Conference on Automated Software Engineering*. Auckland, New Zeeland: IEEE Computer Society, Nov. 2009, pp. 88–99.

[8] "Sofl: simulator of fault localization website," http://swerl.tudelft.nl/bin/view/Main/SOFL, 2011, last accessed Jan. 2012.

[9] É. Piel, A. Gonzalez-Sanchez, H.-G. Gross, and A. J. V. Gemund, "Spectrum-based health monitoring for self-adaptive systems," in *5th IEEE Int'l Conference on Self-Adaptive and Self-Organizing Systems (SASO'11)*. Ann Arbor, USA: IEEE Computer Society, Oct. 2011.

[10] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zeeland, Nov. 2009, pp. 550–554.

[11] "Atlas component framework website," http://swerl.tudelft.nl/bin/view/Main/Atlas, 2010, last accessed Jan. 2012.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[13] *Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band*, International Telecommunications Union, 2001, Recommendation ITU-R M.1371-1.

[14] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Software Testing Verification and Reliability*, vol. 15, pp. 97–133, Jun. 2005.

 ISBN: 978-1-61208-187-8

[15] "Berkeley/stanford recovery-oriented computing website," http://roc.cs.berkeley.edu/, 2005, last accessed Jan. 2012.

[16] "XEROX Model-Based Computing project website," http://www2.parc.com/spl/projects/mbc/, 1997, last accessed Jan. 2012.

[17] G. K. Baah, E. Gray, and M. J. Harrold, "On-line anomaly detection of deployed software: a statistical machine learning approach," in *Proc. of the 3rd International Workshop on Software Quality Assurance*. Portland, Oregon, USA: ACM, Nov. 2006, pp. 70–77.

[18] C. Rabejac, J.-P. Blanquart, and J.-P. Queille, "Executable assertions and timed traces for on-line software error detection," in *Annual Symposium on Fault Tolerant Computing*, Sendai, Japan, Jun. 1996, pp. 138–147.

[19] M. Seltzer and C. Small, "Self-monitoring and self-adapting operating systems," in *Proc. of the Workshop on Hot Topics in Operating Systems*, Cape Cod, Massachusetts, USA, May 1997, pp. 124–129.

[20] Z. Chen, "Service fault localization using probing technology," in *Proceedings of the Conference on Networking, Sensing and Control*, Ft. Lauderdale, Florida, USA, Apr. 2006, pp. 937–942.

[21] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[22] P. Casanova, B. R. Schmerl, D. Garlan, and R. Abreu, "Architecture-based run-time fault diagnosis," in *Proc. of the 5th European Conference on Software Architectures (ECSA'11)*, Essen, Germany, Sep. 2011, pp. 261–277.

 ISBN: 978-1-61208-187-8